Contents

Chapter 1

The Router

1.1 Getting started

1.1.1 Installation

1.1.2 Initial configuration

1.1.3 Failsafe mode

1.2 Configuring OpenWrt

1.2.1 Network

The network configuration is stored in `/etc/config/network` and is divided into interface configurations. Each interface configuration either refers directly to an ethernet/wifi interface (`eth0`, `wl0`, ..) or to a bridge containing multiple interfaces. It looks like this:

```
config interface     "lan"
    option ifname    "eth0"
    option proto     "static"
    option ipaddr    "192.168.1.1"
    option netmask   "255.255.255.0"
    option gateway   "192.168.1.254"
    option dns       "192.168.1.254"
```

`ifname` specifies the Linux interface name. If you want to use bridging on one or more interfaces, set `ifname` to a list of interfaces and add:

```
    option type     "bridge"
```

It is possible to use VLAN tagging on an interface simply by adding the VLAN IDs to it, e.g. `eth0.15`. These can be nested as well. See the switch section for this.

```
config interface
    option ifname    "eth0.15"
    option proto     "none"
```

This sets up a simple static configuration for `eth0`. `proto` specifies the protocol used for the interface. The default image usually provides `'none' 'static'`, `'dhcp'` and `'pppoe'`. Others can be added by installing additional packages.

When using the `'static'` method like in the example, the options `ipaddr` and `netmask` are mandatory, while `gateway` and `dns` are optional. You can specify more than one DNS server, separated with spaces:

```
config interface     "lan"
    option ifname    "eth0"
    option proto     "static"
    ...
    option dns       "192.168.1.254 192.168.1.253" (optional)
```

DHCP currently only accepts `ipaddr` (IP address to request from the server) and `hostname` (client hostname identify as) - both are optional.

```
config interface     "lan"
    option ifname    "eth0"
    option proto     "dhcp"
    option ipaddr    "192.168.1.1" (optional)
    option hostname  "openwrt"     (optional)
```

PPP based protocols (`pppoe`, `pptp`, ...) accept these options:

- username
  The PPP username (usually with PAP authentication)
- password
  The PPP password
- keepalive
  Ping the PPP server (using LCP). The value of this option specifies the maximum number of failed pings before reconnecting. The ping interval defaults to 5, but can be changed by appending ",<interval>" to the keepalive value
- demand
  Use Dial on Demand (value specifies the maximum idle time.
- server: (pptp)
  The remote pptp server IP

For all protocol types, you can also specify the MTU by using the `mtu` option. A sample PPPoE config would look like this:

```
config interface     "lan"
    option ifname    "eth0"
    option proto     "pppoe"
    option username  "username"
    option password  "openwrt"
    option mtu       "1492"      (optional)
```

Setting up static routes

You can set up static routes for a specific interface that will be brought up after the interface is configured.

Simply add a config section like this:

```
config route foo
        option interface  "lan"
        option target     "1.1.1.0"
        option netmask    "255.255.255.0"
        option gateway    "192.168.1.1"
```

The name for the route section is optional, the `interface`, `target` and `gateway` options are mandatory. Leaving out the `netmask` option will turn the route into a host route.

Setting up the switch (broadcom only)

The switch configuration is set by adding a `'switch'` config section. Example:

```
config switch       "eth0"
    option vlan0    "1 2 3 4 5*"
    option vlan1    "0 5"
```

On Broadcom hardware the section name needs to be eth0, as the switch driver does not detect the switch on any other physical device. Every vlan option needs to have the name vlan<n> where <n> is the VLAN number as used in the switch driver. As value it takes a list of ports with these optional suffixes:

- `'*'`: Set the default VLAN (PVID) of the Port to the current VLAN
- `'u'`: Force the port to be untagged
- `'t'`: Force the port to be tagged

The CPU port defaults to tagged, all other ports to untagged. On Broadcom hardware the CPU port is always 5. The other ports may vary with different hardware.

For instance, if you wish to have 3 vlans, like one 3-port switch, 1 port in a DMZ, and another one as your WAN interface, use the following configuration :

```
config switch       "eth0"
    option vlan0    "1 2 3 5*"
    option vlan1    "0 5"
    option vlan2    "4 5"
```

Three interfaces will be automatically created using this switch layout : `eth0.0` (vlan0), `eth0.1` (vlan1) and `eth0.2` (vlan2). You can then assign those interfaces to a custom network configuration name like `lan`, `wan` or `dmz` for instance.

Setting up the switch (swconfig)

swconfig based configurations have a different structure with one extra section per vlan. The example below shows a typical configuration:

```
config 'switch' 'eth0'
        option 'reset' '1'
        option 'enable_vlan' '1'

config 'switch_vlan' 'eth0_1'
        option 'device' 'eth0'
        option 'vlan' '1'
        option 'ports' '0 1 2 3 5t'

config 'switch_vlan' 'eth0_2'
        option 'device' 'eth0'
        option 'vlan' '2'
        option 'ports' '4 5t'
```

Setting up IPv6 connectivity

OpenWrt supports IPv6 connectivity using PPP, Tunnel brokers or static assignment.

If you use PPP, IPv6 will be setup using IP6CP and there is nothing to configure.

To setup an IPv6 tunnel to a tunnel broker, you can install the `6scripts` package and edit the `/etc/config/6tunnel` file and change the settings accordingly :

```
config 6tunnel
        option tnlifname     'sixbone'
        option remoteip4     '1.0.0.1'
        option localip4      '1.0.0.2'
        option localip6      '2001::DEAD::BEEF::1'
```

- 'tnlifname': Set the interface name of the IPv6 in IPv4 tunnel
- 'remoteip4': IP address of the remote end to establish the 6in4 tunnel. This address is given by the tunnel broker
- 'localip4': IP address of your router to establish the 6in4 tunnel. It will usually match your WAN IP address.
- 'localip6': IPv6 address to setup on your tunnel side This address is given by the tunnel broker

Using the same package you can also setup an IPv6 bridged connection:

```
config 6bridge
        option bridge        'br6'
```

By default the script bridges the WAN interface with the LAN interface and uses ebtables to filter anything that is not IPv6 on the bridge. This configuration is particularly useful if your router is not IPv6 ND proxy capable (see: http://www.rfc-archive.org/getrfc.php?rfc=4389).

IPv6 static addressing is also supported using a similar setup as IPv4 but with the ip6 prefixing (when applicable).

```
config interface     "lan"
    option ifname    "eth0"
    option proto     "static"
    option ip6addr   "fe80::200:ff:fe00:0/64"
    option ip6gw     "2001::DEAF:BEE:1"
```

1.2.2 Wireless

The WiFi settings are configured in the file /etc/config/wireless (currently supported on Broadcom, Atheros and mac80211). When booting the router for the first time it should detect your card and create a sample configuration file. By default 'option network lan' is commented. This prevents unsecured sharing of the network over the wireless interface.

Each wireless driver has its own configuration script in /lib/wifi/driver_name.sh which handles driver specific options and configurations. This script is also calling driver specific binaries like wlc for Broadcom, or hostapd and wpa_supplicant for atheros and mac80211.

The reason for using such architecture, is that it abstracts the driver configuration.

Generic Broadcom wireless config:

```
config wifi-device      "wl0"
    option type         "broadcom"
    option channel      "5"

config wifi-iface
    option device       "wl0"
#   option network  lan
    option mode         "ap"
    option ssid         "OpenWrt"
    option hidden       "0"
    option encryption   "none"
```

Generic Atheros wireless config:

```
config wifi-device      "wifi0"
    option type         "atheros"
    option channel      "5"
    option hwmode        "11g"

config wifi-iface
    option device       "wifi0"
#   option network  lan
    option mode         "ap"
    option ssid         "OpenWrt"
    option hidden       "0"
    option encryption   "none"
```

Generic mac80211 wireless config:

```
config wifi-device      "wifi0"
    option type         "mac80211"
    option channel      "5"

config wifi-iface
    option device       "wlan0"
#   option network  lan
    option mode         "ap"
    option ssid         "OpenWrt"
    option hidden       "0"
    option encryption   "none"
```

Generic multi-radio Atheros wireless config:

```
config wifi-device  wifi0
    option type     atheros
```

```
    option channel  1

config wifi-iface
    option device   wifi0
#   option network  lan
    option mode     ap
    option ssid     OpenWrt_private
    option hidden   0
    option encryption none

config wifi-device  wifi1
    option type     atheros
    option channel  11

config wifi-iface
    option device   wifi1
#   option network  lan
    option mode     ap
    option ssid     OpenWrt_public
    option hidden   1
    option encryption none
```

There are two types of config sections in this file. The '`wifi-device`' refers to the physical wifi interface and '`wifi-iface`' configures a virtual interface on top of that (if supported by the driver).

A full outline of the wireless configuration file with description of each field:

```
config wifi-device    wifi device name
    option type       broadcom, atheros, mac80211
    option country    us, uk, fr, de, etc.
    option channel    1-14
    option maxassoc   1-128 (broadcom only)
    option distance   1-n (meters)
    option hwmode     11b, 11g, 11a, 11bg (atheros, mac80211)
    option rxantenna  0,1,2 (atheros, broadcom)
    option txantenna  0,1,2 (atheros, broadcom)
    option txpower  transmission power in dBm

config wifi-iface
    option network  the interface you want wifi to bridge with
    option device   wifi0, wifi1, wifi2, wifiN
    option mode     ap, sta, adhoc, monitor, mesh, or wds
    option txpower  (deprecated) transmission power in dBm
    option ssid     ssid name
    option bssid    bssid address
    option encryption none, wep, psk, psk2, wpa, wpa2
    option key      encryption key
    option key1     key 1
    option key2     key 2
```

```
option key3     key 3
option key4     key 4
option passphrase 0,1
option server   ip address
option port     port
option hidden   0,1
option isolate  0,1         (broadcom)
option doth     0,1         (atheros, broadcom)
option wmm      0,1         (atheros, broadcom)
```

Options for the `wifi-device`:

- `type`
  The driver to use for this interface.

- `country`
  The country code used to determine the regulatory settings.

- `channel`
  The wifi channel (e.g. 1-14, depending on your country setting).

- `maxassoc`
  Optional: Maximum number of associated clients. This feature is supported only on the Broadcom chipsets.

- `distance`
  Optional: Distance between the ap and the furthest client in meters. This feature is supported only on the Atheros chipsets.

- `mode`
  The frequency band (`b`, `g`, `bg`, `a`). This feature is only supported on the Atheros chipsets.

- `diversity`
  Optional: Enable diversity for the Wi-Fi device. This feature is supported only on the Atheros chipsets.

- `rxantenna`
  Optional: Antenna identifier (0, 1 or 2) for reception. This feature is supported by Atheros and some Broadcom chipsets.

- `txantenna`
  Optional: Antenna identifier (0, 1 or 2) for emission. This feature is supported by Atheros and some Broadcom chipsets.

- `txpower` Set the transmission power to be used. The amount is specified in dBm.

Options for the `wifi-iface`:

- `network`
  Selects the interface section from `/etc/config/network` to be used with this interface
- `device`
  Set the wifi device name.
- `mode`
  Operating mode:
  - `ap`
    Access point mode
  - `sta`
    Client mode
  - `adhoc`
    Ad-Hoc mode
  - `monitor`
    Monitor mode
  - `mesh`
    Mesh Point mode (802.11s)
  - `wds`
    WDS point-to-point link
- `ssid` Set the SSID to be used on the wifi device.
- `bssid` Set the BSSID address to be used for wds to set the mac address of the other wds unit.
- `txpower` (Deprecated, set in wifi-device) Set the transmission power to be used. The amount is specified in dBm.
- `encryption`
  Encryption setting. Accepts the following values:
  - `none`
  - `wep`
  - `psk`, `psk2`
    WPA(2) Pre-shared Key
  - `wpa`, `wpa2`
    WPA(2) RADIUS
- `key, key1, key2, key3, key4` (wep, wpa and psk)
  WEP key, WPA key (PSK mode) or the RADIUS shared secret (WPA RADIUS mode)

- `passphrase` (wpa)
  0 treats the wpa psk as a text passphrase; 1 treats wpa psk as encoded passphrase. You can generate an encoded passphrase with the wpa_passphrase utility. This is especially useful if your passphrase contains special characters. This option only works when using mac80211 or atheros type devices.

- `server` (wpa)
  The RADIUS server ip address

- `port` (wpa)
  The RADIUS server port (defaults to 1812)

- `hidden`
  0 broadcasts the ssid; 1 disables broadcasting of the ssid

- `isolate`
  Optional: Isolation is a mode usually set on hotspots that limits the clients to communicate only with the AP and not with other wireless clients. 0 disables ap isolation (default); 1 enables ap isolation.

- `doth`
  Optional: Toggle 802.11h mode. 0 disables 802.11h (default); 1 enables it.

- `wmm`
  Optional: Toggle 802.11e mode. 0 disables 802.11e (default); 1 enables it.

Mesh Point Mesh Point (802.11s) is only supported by some mac80211 drivers. It requires the iw package to be installed to setup mesh links. OpenWrt creates mshN mesh point interfaces. A sample configuration looks like this:

```
config wifi-device      "wlan0"
    option type                "mac80211"
    option channel      "5"

config wifi-iface
    option device       "wlan0"
    option network          lan
    option mode         "mesh"
    option mesh_id     "OpenWrt"
```

Wireless Distribution System WDS is a non-standard mode which will be working between two Broadcom devices for instance but not between a Broadcom and Atheros device.

Unencrypted WDS connections This configuration example shows you how to setup unencrypted WDS connections. We assume that the peer configured as below as the BSSID ca:fe:ba:be:00:01 and the remote WDS endpoint ca:fe:ba:be:00:02 (option bssid field).

```
config wifi-device      "wl0"
    option type                 "broadcom"
    option channel      "5"

config wifi-iface
    option device       "wl0"
    option network          lan
    option mode         "ap"
    option ssid         "OpenWrt"
    option hidden       "0"
    option encryption   "none"

config wifi-iface
    option device       "wl0"
    option network      lan
    option mode         wds
    option ssid         "OpenWrt WDS"
    option bssid        "ca:fe:ba:be:00:02"
```

Encrypted WDS connections It is also possible to encrypt WDS connections. `psk`, `psk2` and `psk+psk2` modes are supported. Configuration below is an example configuration using Pre-Shared-Keys with AES algorithm.

```
config wifi-device  wl0
    option type     broadcom
    option channel  5

config wifi-iface
    option device   "wl0"
    option network  lan
    option mode     ap
    option ssid     "OpenWrt"
    option encryption  psk2
    option key      "<key for clients>"

config wifi-iface
    option device   "wl0"
    option network  lan
    option mode     wds
    option bssid    ca:fe:ba:be:00:02
    option ssid     "OpenWrt WDS"
    option encryption        psk2
    option key      "<psk for WDS>"
```

802.1x configurations OpenWrt supports both 802.1x client and Access Point configurations. 802.1x client is only working with drivers supported by wpa-supplicant. Configuration only supports EAP types TLS, TTLS or PEAP.

EAP-TLS

```
config wifi-iface
    option device         "ath0"
    option network        lan
    option ssid           OpenWrt
    option eap_type       tls
    option ca_cert        "/etc/config/certs/ca.crt"
    option priv_key       "/etc/config/certs/priv.crt"
    option priv_key_pwd   "PKCS#12 passphrase"
```

EAP-PEAP

```
config wifi-iface
    option device         "ath0"
    option network        lan
    option ssid           OpenWrt
    option eap_type       peap
    option ca_cert        "/etc/config/certs/ca.crt"
    option auth           MSCHAPV2
    option identity       username
    option password       password
```

Limitations: There are certain limitations when combining modes. Only the following mode combinations are supported:

- **Broadcom**:
  - 1x `sta`, 0-3x `ap`
  - 1-4x `ap`
  - 1x `adhoc`
  - 1x `monitor`

  WDS links can only be used in pure AP mode and cannot use WEP (except when sharing the settings with the master interface, which is done automatically).
- **Atheros**:
  - 1x `sta`, 0-Nx `ap`
  - 1-Nx `ap`
  - 1x `adhoc`

  N is the maximum number of VAPs that the module allows, it defaults to 4, but can be changed by loading the module with the maxvaps=N parameter.

Adding a new driver configuration Since we currently only support thread different wireless drivers : Broadcom, Atheros and mac80211, you might be interested in adding support for another driver like Ralink RT2x00, Texas Instruments ACX100/111.

The driver specific script should be placed in `/lib/wifi/<driver>.sh` and has to include several functions providing :

- detection of the driver presence
- enabling/disabling the wifi interface(s)
- configuration reading and setting
- third-party programs calling (nas, supplicant)

Each driver script should append the driver to a global DRIVERS variable :

```
append DRIVERS "driver name"
```

`scan_<driver>` This function will parse the `/etc/config/wireless` and make sure there are no configuration incompatibilities, like enabling hidden SSIDS with ad-hoc mode for instance. This can be more complex if your driver supports a lof of configuration options. It does not change the state of the interface.

Example:

```
scan_dummy() {
        local device="$1"

        config_get vifs "$device" vifs
        for vif in $vifs; do
                # check config consistency for wifi-iface sections
        done
        # check mode combination
}
```

`enable_<driver>` This function will bring up the wifi device and optionally create application specific configuration files, e.g. for the WPA authenticator or supplicant.

Example:

```
enable_dummy() {
        local device="$1"

        config_get vifs "$device" vifs
        for vif in $vifs; do
                # bring up virtual interface belonging to
                # the wifi-device "$device"
        done
}
```

`disable_<driver>` This function will bring down the wifi device and all its virtual interfaces (if supported).

Example:

```
disable_dummy() {
        local device="$1"

        # bring down virtual interfaces belonging to
        # "$device" regardless of whether they are
        # configured or not. Don't rely on the vifs
        # variable at this point
}
```

`detect_<driver>` This function looks for interfaces that are usable with the driver. Template config sections for new devices should be written to stdout. Must check for already existing config sections belonging to the interfaces before creating new templates.

Example:

```
detect_dummy() {
        [ wifi-device = "$(config_get dummydev type)" ] && return 0
        cat <<EOF
config wifi-device dummydev
        option type dummy
        # REMOVE THIS LINE TO ENABLE WIFI:
        option disabled 1

config wifi-iface
        option device dummydev
        option mode ap
        option ssid OpenWrt
EOF
}
```

1.3 Advanced configuration

Structure of the configuration files

The config files are divided into sections and options/values.

Every section has a type, but does not necessarily have a name. Every option has a name and a value and is assigned to the section it was written under.

Syntax:

```
config      <type> ["<name>"]      # Section
    option  <name> "<value>"       # Option
```

Every parameter needs to be a single string and is formatted exactly like a parameter for a shell function. The same rules for Quoting and special characters also apply, as it is parsed by the shell.

Parsing configuration files in custom scripts

To be able to load configuration files, you need to include the common functions with:

```
. /lib/functions.sh
```

Then you can use `config_load <name>` to load config files. The function first checks for <*name*> as absolute filename and falls back to loading it from `/etc/config` (which is the most common way of using it).

If you want to use special callbacks for sections and/or options, you need to define the following shell functions before running `config_load` (after including `/lib/functions.sh`):

```
config_cb() {
    local type="$1"
    local name="$2"
    # commands to be run for every section
}

option_cb() {
    # commands to be run for every option
}
```

You can also alter `option_cb` from `config_cb` based on the section type. This allows you to process every single config section based on its type individually.

`config_cb` is run every time a new section starts (before options are being processed). You can access the last section through the `CONFIG_SECTION` variable. Also an extra call to `config_cb` (without a new section) is generated after `config_load` is done. That allows you to process sections both before and after all options were processed.

Another way of iterating on config sections is using the `config_foreach` command.

Syntax:

```
config_foreach <function name> [<sectiontype>] [<arguments...>]
```

This command will run the supplied function for every single config section in the currently loaded config. The section name will be passed to the function as argument 1. If the section type is added to the command line, the function will only be called for sections of the given type.

You can access already processed options with the `config_get` command Syntax:

```
# print the value of the option
config_get <section> <option>

# store the value inside the variable
config_get <variable> <section> <option>
```

In busybox ash the three-option `config_get` is faster, because it does not result in an extra fork, so it is the preferred way.

Additionally you can also modify or add options to sections by using the `config_set` command.

Syntax:

```
config_set <section> <option> <value>
```

If a config section is unnamed, an automatically generated name will be assigned internally, e.g. `cfg1`, `cfg2`, ...

While it is possible, using unnamed sections through these autogenerated names is strongly discouraged. Use callbacks or `config_foreach` instead.

1.3.1 Hotplug

1.3.2 Init scripts

Because OpenWrt uses its own init script system, all init scripts must be installed as `/etc/init.d/`*name* use `/etc/rc.common` as a wrapper.

Example: `/etc/init.d/httpd`

```
#!/bin/sh /etc/rc.common
# Copyright (C) 2006 OpenWrt.org

START=50
start() {
    [ -d /www ] && httpd -p 80 -h /www -r OpenWrt
}

stop() {
    killall httpd
}
```

as you can see, the script does not actually parse the command line arguments itself. This is done by the wrapper script `/etc/rc.common`.

`start()` and `stop()` are the basic functions, which almost any init script should provide. `start()` is called when the user runs `/etc/init.d/httpd start` or (if the script is enabled and does not override this behavior) at system boot time.

Enabling and disabling init scripts is done by running `/etc/init.d/name enable` or `/etc/init.d/name disable`. This creates or removes symbolic links to the init script in `/etc/rc.d`, which is processed by `/etc/init.d/rcS` at boot time.

The order in which these scripts are run is defined in the variable `START` in the init script. Changing it requires running `/etc/init.d/name enable` again.

You can also override these standard init script functions:

- `boot()`
  Commands to be run at boot time. Defaults to `start()`
- `restart()`
  Restart your service. Defaults to `stop(); start()`
- `reload()`
  Reload the configuration files for your service. Defaults to `restart()`

You can also add custom commands by creating the appropriate functions and referencing them in the `EXTRA_COMMANDS` variable. Helptext is added in `EXTRA_HELP`.

Example:

```
status() {
    # print the status info
}

EXTRA_COMMANDS="status"
EXTRA_HELP="        status  Print the status of the service"
```

1.3.3 Network scripts

Using the network scripts

To be able to access the network functions, you need to include the necessary shell scripts by running:

```
. /lib/functions.sh      # common functions
include /lib/network     # include /lib/network/*.sh
scan_interfaces          # read and parse the network config
```

Some protocols, such as PPP might change the configured interface names at run time (e.g. `eth0` => `ppp0` for PPPoE). That's why you have to run `scan_interfaces` instead of reading the values from the config directly. After running `scan_interfaces`, the `'ifname'` option will always contain the effective interface name (which is used for IP traffic) and if the physical device name differs from it, it will be stored in the `'device'` option. That means that running `config_get lan ifname` after `scan_interfaces` might not return the same result as running it before.

After running `scan_interfaces`, the following functions are available:

- `find_config` *interface*
  looks for a network configuration that includes the specified network interface.
- `setup_interface` *interface [config] [protocol]*
  will set up the specified interface, optionally overriding the network configuration name or the protocol that it uses.

Writing protocol handlers

You can add custom protocol handlers (e.g: PPPoE, PPPoA, ATM, PPTP ...) by adding shell scripts to `/lib/network`. They provide the following two shell functions:

```
scan_<protocolname>() {
    local config="$1"
    # change the interface names if necessary
}

setup_interface_<protocolname>() {
    local interface="$1"
    local config="$2"
    # set up the interface
}
```

`scan_`*protocolname* is optional and only necessary if your protocol uses a custom device, e.g. a tunnel or a PPP device.

Chapter 2

Development issues

2.1 The build system

One of the biggest challenges to getting started with embedded devices is that you cannot just install a copy of Linux and expect to be able to compile a firmware. Even if you did remember to install a compiler and every development tool offered, you still would not have the basic set of tools needed to produce a firmware image. The embedded device represents an entirely new hardware platform, which is most of the time incompatible with the hardware on your development machine, so in a process called cross compiling you need to produce a new compiler capable of generating code for your embedded platform, and then use it to compile a basic Linux distribution to run on your device.

The process of creating a cross compiler can be tricky, it is not something that is regularly attempted and so there is a certain amount of mystery and black magic associated with it. In many cases when you are dealing with embedded devices you will be provided with a binary copy of a compiler and basic libraries rather than instructions for creating your own – it is a time saving step but at the same time often means you will be using a rather dated set of tools. Likewise, it is also common to be provided with a patched copy of the Linux kernel from the board or chip vendor, but this is also dated and it can be difficult to spot exactly what has been modified to make the kernel run on the embedded platform.

2.1.1 Building an image

OpenWrt takes a different approach to building a firmware; downloading, patching and compiling everything from scratch, including the cross compiler. To put it in simpler terms, OpenWrt does not contain any executables or even sources, it is an automated system for downloading the sources, patching them to work with the given platform and compiling them correctly for that platform. What this means is that just by changing the template, you can change any step in the process.

As an example, if a new kernel is released, a simple change to one of the Makefiles will download the latest kernel, patch it to run on the embedded platform and

produce a new firmware image – there is no work to be done trying to track down an unmodified copy of the existing kernel to see what changes had been made, the patches are already provided and the process ends up almost completely transparent. This does not just apply to the kernel, but to anything included with OpenWrt – It is this one simple understated concept which is what allows OpenWrt to stay on the bleeding edge with the latest compilers, latest kernels and latest applications.

So let's take a look at OpenWrt and see how this all works.

Download OpenWrt

OpenWrt can be downloaded via subversion using the following command:

```
$ svn checkout svn://svn.openwrt.org/openwrt/trunk openwrt-trunk
```

Additionally, there is a trac interface on https://dev.openwrt.org/ which can be used to monitor svn commits and browse the source repository.

The directory structure

There are four key directories in the base:

- `tools`
- `toolchain`
- `package`
- `target`

`tools` and `toolchain` refer to common tools which will be used to build the firmware image, the compiler, and the C library. The result of this is three new directories, `build_dir/host`, which is a temporary directory for building the target independent tools, `build_dir/toolchain-`*`<arch>`*`*` which is used for building the toolchain for a specific architecture, and `staging_dir/toolchain-`*`<arch>`*`*` where the resulting toolchain is installed. You will not need to do anything with the toolchain directory unless you intend to add a new version of one of the components above.

- `build_dir/host`
- `build_dir/toolchain-`*`<arch>`*`*`

`package` is for exactly that – packages. In an OpenWrt firmware, almost everything is an `.ipk`, a software package which can be added to the firmware to provide new features or removed to save space. Note that packages are also maintained outside of the main trunk and can be obtained from subversion using the package feeds system:

```
$ ./scripts/feeds update
```

Those packages can be used to extend the functionality of the build system and need to be symlinked into the main trunk. Once you do that, the packages will show up in the menu for configuration. You would do something like this:

```
$ ./scripts/feeds search nmap
Search results in feed 'packages':
nmap       Network exploration and/or security auditing utility

$ ./scripts/feeds install nmap
```

To include all packages, issue the following command:

```
$ make package/symlinks
```

`target` refers to the embedded platform, this contains items which are specific to a specific embedded platform. Of particular interest here is the "`target/linux`" directory which is broken down by platform <*arch*> and contains the patches to the kernel, profile config, for a particular platform. There's also the "`target/image`" directory which describes how to package a firmware for a specific platform.

Both the target and package steps will use the directory "`build_dir/<arch>`" as a temporary directory for compiling. Additionally, anything downloaded by the toolchain, target or package steps will be placed in the "`dl`" directory.

- `build_dir/<arch>`
- `dl`

Building OpenWrt

While the OpenWrt build environment was intended mostly for developers, it also has to be simple enough that an inexperienced end user can easily build his or her own customized firmware.

Running the command "`make menuconfig`" will bring up OpenWrt's configuration menu screen, through this menu you can select which platform you're targeting, which versions of the toolchain you want to use to build and what packages you want to install into the firmware image. Note that it will also check to make sure you have the basic dependencies for it to run correctly. If that fails, you will need to install some more tools in your local environment before you can begin.

Similar to the linux kernel config, almost every option has three choices, `y/m/n` which are represented as follows:

- `<*>` (pressing y)
  This will be included in the firmware image

- <M> (pressing m)
  This will be compiled but not included (for later install)
- < > (pressing n)
  This will not be compiled

After you've finished with the menu configuration, exit and when prompted, save your configuration changes.

If you want, you can also modify the kernel config for the selected target system. simply run "`make kernel_menuconfig`" and the build system will unpack the kernel sources (if necessary), run menuconfig inside of the kernel tree, and then copy the kernel config to `target/linux/<platform>/config` so that it is preserved over "`make clean`" calls.

To begin compiling the firmware, type "`make`". By default OpenWrt will only display a high level overview of the compile process and not each individual command.

Example:

```
make[2] toolchain/install
make[3] -C toolchain install
make[2] target/compile
make[3] -C target compile
make[4] -C target/utils prepare

[...]
```

This makes it easier to monitor which step it's actually compiling and reduces the amount of noise caused by the compile output. To see the full output, run the command "`make V=99`".

During the build process, buildroot will download all sources to the "`dl`" directory and will start patching and compiling them in the "`build_dir/<arch>`" directory. When finished, the resulting firmware will be in the "`bin`" directory and packages will be in the "`bin/packages`" directory.

2.1.2 Creating packages

One of the things that we've attempted to do with OpenWrt's template system is make it incredibly easy to port software to OpenWrt. If you look at a typical package directory in OpenWrt you'll find several things:

- `package/<name>/Makefile`
- `package/<name>/patches`
- `package/<name>/files`

The patches directory is optional and typically contains bug fixes or optimizations to reduce the size of the executable. The package makefile is the important item, provides the steps actually needed to download and compile the package.

The files directory is also optional and typicall contains package specific startup scripts or default configuration files that can be used out of the box with OpenWrt.

Looking at one of the package makefiles, you'd hardly recognize it as a makefile. Through what can only be described as blatant disregard and abuse of the traditional make format, the makefile has been transformed into an object oriented template which simplifies the entire ordeal.

Here for example, is `package/bridge/Makefile`:

```
1
2  include $(TOPDIR)/rules.mk
3
4  PKG_NAME:=bridge
5  PKG_VERSION:=1.0.6
6  PKG_RELEASE:=1
7
8  PKG_SOURCE:=bridge-utils-$(PKG_VERSION).tar.gz
9  PKG_SOURCE_URL:=@SF/bridge
10 PKG_MD5SUM:=9b7dc52656f5cbec846a7ba3299f73bd
11 PKG_CAT:=zcat
12
13 PKG_BUILD_DIR:=$(BUILD_DIR)/bridge-utils-$(PKG_VERSION)
14
15 include $(INCLUDE_DIR)/package.mk
16
17 define Package/bridge
18   SECTION:=net
19   CATEGORY:=Base system
20   TITLE:=Ethernet bridging configuration utility
21   URL:=http://bridge.sourceforge.net/
22 endef
23
24 define Package/bridge/description
25   Manage ethernet bridging:
26   a way to connect networks together to form a larger network.
27 endef
28
29 define Build/Configure
30     $(call Build/Configure/Default, \
31         --with-linux-headers="$(LINUX_DIR)" \
32     )
33 endef
34
35 define Package/bridge/install
36     $(INSTALL_DIR) $(1)/usr/sbin
```

```
37      $(INSTALL_BIN) $(PKG_BUILD_DIR)/brctl/brctl $(1)/usr/sbin/
38  endef
39
40  $(eval $(call BuildPackage,bridge))
```

As you can see, there's not much work to be done; everything is hidden in other makefiles and abstracted to the point where you only need to specify a few variables.

- `PKG_NAME`
  The name of the package, as seen via menuconfig and ipkg
- `PKG_VERSION`
  The upstream version number that we are downloading
- `PKG_RELEASE`
  The version of this package Makefile
- `PKG_SOURCE`
  The filename of the original sources
- `PKG_SOURCE_URL`
  Where to download the sources from (no trailing slash), you can add multiple download sources by separating them with a
  and a carriage return.
- `PKG_MD5SUM`
  A checksum to validate the download
- `PKG_CAT`
  How to decompress the sources (zcat, bzcat, unzip)
- `PKG_BUILD_DIR`
  Where to compile the package

The `PKG_*` variables define where to download the package from; `@SF` is a special keyword for downloading packages from sourceforge. There is also another keyword of `@GNU` for grabbing GNU source releases. If any of the above mentionned download source fails, the OpenWrt mirrors will be used as source.

The md5sum (if present) is used to verify the package was downloaded correctly and `PKG_BUILD_DIR` defines where to find the package after the sources are uncompressed into `$(BUILD_DIR)`.

At the bottom of the file is where the real magic happens, "BuildPackage" is a macro set up by the earlier include statements. BuildPackage only takes one argument directly – the name of the package to be built, in this case "`bridge`". All other information is taken from the define blocks. This is a way of providing a level of verbosity, it's inherently clear what the contents of the `description` template in `Package/bridge` is, which wouldn't be the case if we passed this information directly as the Nth argument to `BuildPackage`.

`BuildPackage` uses the following defines:

`Package/<name>:`
<name> matches the argument passed to buildroot, this describes the package the menuconfig and ipkg entries. Within `Package/`*<name>* you can define the following variables:

- `SECTION`
  The section of package (currently unused)
- `CATEGORY`
  Which menu it appears in menuconfig: Network, Sound, Utilities, Multimedia ...
- `TITLE`
  A short description of the package
- `URL`
  Where to find the original software
- `MAINTAINER` (optional)
  Who to contact concerning the package
- `DEPENDS` (optional)
  Which packages must be built/installed before this package. To reference a dependency defined in the same Makefile, use *<dependency name>*. If defined as an external package, use *+<dependency name>*. For a kernel version dependency use: *@LINUX_2_<minor version>*
- `BUILDONLY` (optional)
  Set this option to 1 if you do NOT want your package to appear in menuconfig. This is useful for packages which are only used as build dependencies.

`Package/<name>/conffiles` **(optional):**
A list of config files installed by this package, one file per line.

`Build/Prepare` **(optional):**
A set of commands to unpack and patch the sources. You may safely leave this undefined.

`Build/Configure` **(optional):**
You can leave this undefined if the source doesn't use configure or has a normal config script, otherwise you can put your own commands here or use "`$(call Build/Configure/Default,`*<first list of arguments, second list>*`)`" as above to pass in additional arguments for a standard configure script. The first list of arguments will be passed to the configure script like that: `-arg 1 -arg 2`. The second list contains arguments that should be defined before running the configure script such as autoconf or compiler specific variables.

To make it easier to modify the configure command line, you can either extend or completely override the following variables:

- `CONFIGURE_ARGS`
  Contains all command line arguments (format: `-arg 1 -arg 2`)

- `CONFIGURE_VARS`
  Contains all environment variables that are passed to ./configure (format: `NAME="value"`)

`Build/Compile` **(optional):**
How to compile the source; in most cases you should leave this undefined.

As with `Build/Configure` there are two variables that allow you to override the make command line environment variables and flags:

- `MAKE_FLAGS`
  Contains all command line arguments (typically variable overrides like `NAME="value"`
- `MAKE_VARS`
  Contains all environment variables that are passed to the make command

`Build/InstallDev` **(optional):**
If your package provides a library that needs to be made available to other packages, you can use the `Build/InstallDev` template to copy it into the staging directory which is used to collect all files that other packages might depend on at build time. When it is called by the build system, two parameters are passed to it. `$(1)` points to the regular staging dir, typically `staging_dir/`*`ARCH`*, while `$(2)` points to `staging_dir/host`. The host staging dir is only used for binaries, which are to be executed or linked against on the host and its `bin/` subdirectory is included in the `PATH` which is passed down to the build system processes. Please use `$(1)` and `$(2)` here instead of the build system variables `$(STAGING_DIR)` and `$(STAGING_DIR_HOST)`, because the build system behavior when staging libraries might change in the future to include automatic uninstallation.

`Package/<name>/install:`
A set of commands to copy files out of the compiled source and into the ipkg which is represented by the `$(1)` directory. Note that there are currently 4 defined install macros:

- `INSTALL_DIR`
  install -d -m0755
- `INSTALL_BIN`
  install -m0755
- `INSTALL_DATA`
  install -m0644
- `INSTALL_CONF`
  install -m0600

The reason that some of the defines are prefixed by "`Package/`*`<name>`*" and others are simply "`Build`" is because of the possibility of generating multiple packages from a single source. OpenWrt works under the assumption of one source per package Makefile, but you can split that source into as many packages as desired. Since you only need to compile the sources once, there's one global

set of "`Build`" defines, but you can add as many "Package/<name>" defines as you want by adding extra calls to `BuildPackage` – see the dropbear package for an example.

After you have created your `package/<name>/Makefile`, the new package will automatically show in the menu the next time you run "make menuconfig" and if selected will be built automatically the next time "`make`" is run.

2.1.3 Creating binary packages

You might want to create binary packages and include them in the resulting images as packages. To do so, you can use the following template, which basically sets to nothing the Configure and Compile templates.

```
1
2  include $(TOPDIR)/rules.mk
3
4  PKG_NAME:=binpkg
5  PKG_VERSION:=1.0
6  PKG_RELEASE:=1
7
8  PKG_SOURCE:=binpkg-$(PKG_VERSION).tar.gz
9  PKG_SOURCE_URL:=http://server
10 PKG_MD5SUM:=9b7dc52656f5cbec846a7ba3299f73bd
11 PKG_CAT:=zcat
12
13 include $(INCLUDE_DIR)/package.mk
14
15 define Package/binpkg
16   SECTION:=net
17   CATEGORY:=Network
18   TITLE:=Binary package
19 endef
20
21 define Package/bridge/description
22   Binary package
23 endef
24
25 define Build/Configure
26 endef
27
28 define Build/Compile
29 endef
30
31 define Package/bridge/install
32     $(INSTALL_DIR) $(1)/usr/sbin
33     $(INSTALL_BIN) $(PKG_BUILD_DIR)/* $(1)/usr/sbin/
34 endef
```

```
35
36 $(eval $(call BuildPackage,bridge))
```

Provided that the tarball which contains the binaries reflects the final directory layout (/usr, /lib ...), it becomes very easy to get your package look like one build from sources.

Note that using the same technique, you can easily create binary pcakages for your proprietary kernel modules as well.

2.1.4 Creating kernel modules packages

The OpenWrt distribution makes the distinction between two kind of kernel modules, those coming along with the mainline kernel, and the others available as a separate project. We will see later that a common template is used for both of them.

For kernel modules that are part of the mainline kernel source, the makefiles are located in *package/kernel/modules/*.mk* and they appear under the section "Kernel modules"

For external kernel modules, you can add them to the build system just like if they were software packages by defining a KernelPackage section in the package makefile.

Here for instance the Makefile for the I2C subsytem kernel modules :

```
1
2  I2CMENU:=I2C Bus
3
4  define KernelPackage/i2c-core
5    TITLE:=I2C support
6    DESCRIPTION:=Kernel modules for i2c support
7    SUBMENU:=$(I2CMENU)
8    KCONFIG:=CONFIG_I2C_CORE CONFIG_I2C_DEV
9    FILES:=$(MODULES_DIR)/kernel/drivers/i2c/*.$(LINUX_KMOD_SUFFIX)
10   AUTOLOAD:=$(call AutoLoad,50,i2c-core i2c-dev)
11 endef
12 $(eval $(call KernelPackage,i2c-core))
```

To group kernel modules under a common description in menuconfig, you might want to define a *<description>MENU* variable on top of the kernel modules makefile.

- `TITLE`
  The name of the module as seen via menuconfig
- `DESCRIPTION`
  The description as seen via help in menuconfig

- SUBMENU
  The sub menu under which this package will be seen
- KCONFIG
  Kernel configuration option dependency. For external modules, remove it.
- FILES
  Files you want to inlude to this kernel module package, separate with spaces.
- AUTOLOAD
  Modules that will be loaded automatically on boot, the order you write them is the order they would be loaded.

After you have created your package/kernel/modules/<*name*>.mk, the new kernel modules package will automatically show in the menu under "Kernel modules" next time you run "make menuconfig" and if selected will be built automatically the next time "make" is run.

2.1.5 Conventions

There are a couple conventions to follow regarding packages:

- files
  1. configuration files follow the convention
     <*name*>.conf
  2. init files follow the convention
     <*name*>.init
- patches
  1. patches are numerically prefixed and named related to what they do

2.1.6 Troubleshooting

If you find your package doesn't show up in menuconfig, try the following command to see if you get the correct description:

```
TOPDIR=$PWD make -C package/<name> DUMP=1 V=99
```

If you're just having trouble getting your package to compile, there's a few shortcuts you can take. Instead of waiting for make to get to your package, you can run one of the following:

- make package/<*name*>/clean V=99
- make package/<*name*>/install V=99

Another nice trick is that if the source directory under `build_dir/<arch>` is newer than the package directory, it won't clobber it by unpacking the sources again. If you were working on a patch you could simply edit the sources under the `build_dir/<arch>/<source>` directory and run the install command above, when satisfied, copy the patched sources elsewhere and diff them with the unpatched sources. A warning though - if you go modify anything under `package/<name>` it will remove the old sources and unpack a fresh copy.

Other useful targets include:

- `make package/<name>/prepare V=99`
- `make package/<name>/compile V=99`
- `make package/<name>/configure V=99`

2.1.7 Using build environments

OpenWrt provides a means of building images for multiple configurations which can use multiple targets in one single checkout. These *environments* store a copy of the .config file generated by `make menuconfig` and the contents of the `./files` folder. The script `./scripts/env` is used to manage these environments, it uses `git` (which needs to be installed on your system) as backend for version control.

The command

```
./scripts/env help
```

produces a short help text with a list of commands.

To create a new environment named `current`, run the following command

```
./scripts/env new current
```

This will move your `.config` file and `./files` (if it exists) to the `env/` subdirectory and create symlinks in the base folder.

After running make menuconfig or changing things in files/, your current state will differ from what has been saved before. To show these changes, use:

```
./scripts/env diff
```

If you want to save these changes, run:

```
./scripts/env save
```

If you want to revert your changes to the previously saved copy, run:

```
./scripts/env revert
```

If you want, you can now create a second environment using the `new` command. It will ask you whether you want to make it a clone of the current environment (e.g. for minor changes) or if you want to start with a clean version (e.g. for selecting a new target).

To switch to a different environment (e.g. `test1`), use:

```
./scripts/env switch test1
```

To rename the current branch to a new name (e.g. `test2`), use:

```
./scripts/env rename test2
```

If you want to get rid of environment switching and keep everything in the base directory again, use:

```
./scripts/env clear
```

2.2 Extra tools

2.2.1 Image Builder

2.2.2 SDK

2.3 Working with OpenWrt

The following section gives some tips and tricks on how to use efficiently OpenWrt on a regular basis and for daily work.

2.3.1 Compiling/recompiling components

The buildroot allows you to recompile the full environment or only parts of it like the toolchain, the kernel modules, the kernel or some packages.

For instance if you want to recompile the toolchain after you made any change to it issue the following command:

```
make toolchain/{clean,compile,install}
```

Which will clean, compile and install the toolchain. The command actually expands to the following:

```
make[1] toolchain/clean
make[2] -C toolchain/kernel-headers clean
make[2] -C toolchain/binutils clean
make[2] -C toolchain/gcc clean
make[2] -C toolchain/uClibc clean        (glibc or eglibc when chosen)
```

Of course, you could only choose to recompile one or several of the toolchain components (binutils, kernel-headers gcc, C library) individually.

The exact same idea works for packages:

```
make package/busybox/{clean,compile,install}
```

will clean, compile and install busybox (if selected to be installed on the final rootfs).

Supposing that you made changes to the Linux kernel, but do not want to recompile everything, you can recompile only the kernel modules by issuing:

```
make target/linux/compile
```

To recompile the static part of the kernel use the following command:

```
make target/linux/install
```

2.3.2 Using quilt inside OpenWrt

OpenWrt integrates quilt in order to ease the package, kernel and toolchain patches maintenance when migrating over new versions of the software.

Quilt intends to replace an old workflow, where you would download the new source file, create an original copy of it, an a working copy, then try to apply by hand old patches and resolve conflicts manually. Additionnaly, using quilt allows you to update and fold patches into other patches easily.

Quilt is used by default to apply Linux kernel patches, but not for the other components (toolchain and packages).

Using quilt with kernel patches

Assuming that you have everything setup for your new kernel version:

- `LINUX_VERSION` set in the target Makefile
- config-2.6.x.y existing
- patches-2.6.x.y containing the previous patches

Some patches are likely to fail since the vanilla kernel we are patching received modifications so some hunks of the patches are no longer applying. We will use quilt to get them applying cleanly again. Follow this procedure whenever you want to upgrade the kernel using previous patches:

1. make target/linux/clean (removes the old version)
2. make target/linux/compile (uncompress the kernel and try to apply patches)

3. if patches failed to apply:
4. cd build_dir/linux-target/linux-2.6.x.y
5. quilt push -a (to apply patches where quilt stopped)
6. quilt push -f (to force applying patches)
7. edit .rej files, apply the necessary changes to the files
8. remove .rej files
9. quilt refresh
10. repeat operation 3 and following until all patches have been applied
11. when all patches did apply cleanly: make target/linux/refresh

Note that generic (target/linux/generic-2.6/linux-2.6.x/) patches can be found in `build_dir/linux-target/linux-2.6.x.y/patches/generic` and platform specific patches in `build_dir/linux-target/linux-2.6.x.y/patches/platform`.

Using quilt with packages

As we mentionned earlier, quilt is enabled by default for kernel patches, but not for packages. If you want to use quilt in the same way, you should set the QUILT environment variable to 1, e.g:

```
make package/busybox/{clean,compile} QUILT=1
```

Will generate the patch series file and allow you to update patches just like we described before in the kernel case. Note that once all patches apply cleanly you should refresh them as well using the following command:

```
make package/busybox/refresh QUILT=1
```

2.4 Adding platform support

Linux is now one of the most widespread operating system for embedded devices due to its openess as well as the wide variety of platforms it can run on. Many manufacturer actually use it in firmware you can find on many devices: DVB-T decoders, routers, print servers, DVD players ... Most of the time the stock firmware is not really open to the consumer, even if it uses open source software.

You might be interested in running a Linux based firmware for your router for various reasons: extending the use of a network protocol (such as IPv6), having new features, new piece of software inside, or for security reasons. A fully open-source firmware is de-facto needed for such applications, since you want to be free to use this or that version of a particular reason, be able to correct a particular bug. Few manufacturers do ship their routers with a Sample Development Kit,

that would allow you to create your own and custom firmware and most of the time, when they do, you will most likely not be able to complete the firmware creation process.

This is one of the reasons why OpenWrt and other firmware exists: providing a version independent, and tools independent firmware, that can be run on various platforms, known to be running Linux originally.

2.4.1 Which Operating System does this device run?

There is a lot of methods to ensure your device is running Linux. Some of them do need your router to be unscrewed and open, some can be done by probing the device using its external network interfaces.

Operating System fingerprinting and port scanning

A large bunch of tools over the Internet exists in order to let you do OS fingerprinting, we will show here an example using **nmap**:

```
nmap -P0 -O <IP address>
Starting Nmap 4.20 ( http://insecure.org ) at 2007-01-08 11:05 CET
Interesting ports on 192.168.2.1:
Not shown: 1693 closed ports
PORT   STATE SERVICE
22/tcp open  ssh
23/tcp open  telnet
53/tcp open  domain
80/tcp open  http
MAC Address: 00:13:xx:xx:xx:xx (Cisco-Linksys)
Device type: broadband router
Running: Linksys embedded
OS details: Linksys WRT54GS v4 running OpenWrt w/Linux kernel 2.4.30
Network Distance: 1 hop
```

nmap is able to report whether your device uses a Linux TCP/IP stack, and if so, will show you which Linux kernel version is probably runs. This report is quite reliable and it can make the distinction between BSD and Linux TCP/IP stacks and others.

Using the same tool, you can also do port scanning and service version discovery. For instance, the following command will report which IP-based services are running on the device, and which version of the service is being used:

```
nmap -P0 -sV <IP address>
Starting Nmap 4.20 ( http://insecure.org ) at 2007-01-08 11:06 CET
Interesting ports on 192.168.2.1:
Not shown: 1693 closed ports
PORT   STATE SERVICE VERSION
```

```
22/tcp open  ssh     Dropbear sshd 0.48 (protocol 2.0)
23/tcp open  telnet  Busybox telnetd
53/tcp open  domain  ISC Bind dnsmasq-2.35
80/tcp open  http    OpenWrt BusyBox httpd
MAC Address: 00:13:xx:xx:xx:xx (Cisco-Linksys)
Service Info: Device: WAP
```

The web server version, if identified, can be determining in knowing the Operating System. For instance, the **BOA** web server is typical from devices running an open-source Unix or Unix-like.

Wireless Communications Fingerprinting

Although this method is not really known and widespread, using a wireless scanner to discover which OS your router or Access Point run can be used. We do not have a clear example of how this could be achieved, but you will have to monitor raw 802.11 frames and compare them to a very similar device running a Linux based firmware.

Web server security exploits

The Linksys WRT54G was originally hacked by using a "ping bug" discovered in the web interface. This tip has not been fixed for months by Linksys, allowing people to enable the "boot_wait" helper process via the web interface. Many web servers used in firmwares are open source web server, thus allowing the code to be audited to find an exploit. Once you know the web server version that runs on your device, by using **nmap -sV** or so, you might be interested in using exploits to reach shell access on your device.

Native Telnet/SSH access

Some firmwares might have restricted or unrestricted Telnet/SSH access, if so, try to log in with the web interface login/password and see if you can type in some commands. This is actually the case for some Broadcom BCM963xx based firmwares such as the one in Neuf/Cegetel ISP routers, Club-Internet ISP CI-Box and many others. Some commands, like **cat** might be left here and be used to determine the Linux kernel version.

Analysing a binary firmware image

You are very likely to find a firmware binary image on the manufacturer website, even if your device runs a proprietary operating system. If so, you can download it and use an hexadecimal editor to find printable words such as **vmlinux**, **linux**, **ramdisk**, **mtd** and others.

Some Unix tools like **hexdump** or **strings** can be used to analyse the firmware. Below there is an example with a binary firmware found other the Internet:

```
hexdump -C <binary image.extension> | less (more)
00000000  46 49 52 45 32 2e 35 2e  30 00 00 00 00 00 00 00  |FIRE2.5.0.......|
00000010  00 00 00 00 31 2e 30 2e  30 00 00 00 00 00 00 00  |....1.0.0.......|
00000020  00 00 00 00 00 00 00 38  00 43 36 29 00 0a e6 dc  |.......8.C6)..??|
00000030  54 49 44 45 92 89 54 66  1f 8b 08 08 f8 10 68 42  |TIDE..Tf....?.hB|
00000040  02 03 72 61 6d 64 69 73  6b 00 ec 7d 09 bc d5 d3  |..ramdisk.?}.???|
00000050  da ff f3 9b f7 39 7b ef  73 f6 19 3b 53 67 ea 44  |???.?9{?s?.;Sg?D|
```

Scroll over the firmware to find printable words that can be significant.

Amount of flash memory

Linux can hardly fit in a 2MB flash device, once you have opened the device and located the flash chip, try to find its characteristics on the Internet. If your flash chip is a 2MB or less device, your device is most likely to run a proprietary OS such as WindRiver VxWorks, or a custom manufacturer OS like Zyxel ZynOS.

OpenWrt does not currently run on devices which have 2MB or less of flash memory. This limitation will probably not be worked around since those devices are most of the time micro-routers, or Wireless Access Points, which are not the main OpenWrt target.

Pluging a serial port

By using a serial port and a level shifter, you may reach the console that is being shown by the device for debugging or flashing purposes. By analysing the output of this device, you can easily notice if the device uses a Linux kernel or something different.

2.4.2 Finding and using the manufacturer SDK

Once you are sure your device run a Linux based firmware, you will be able to start hacking on it. If the manufacturer respected the GPL, it will have released a Sample Development Kit with the device.

GPL violations

Some manufacturers do release a Linux based binary firmware, with no sources at all. The first step before doing anything is to read the license coming with your device, then write them about this lack of Open Source code. If the manufacturer answers you they do not have to release a SDK containing Open Source software, then we recommend you get in touch with the gpl-violations.org community.

You will find below a sample letter that can be sent to the manufacturer:

> Miss, Mister,
>
> I am using a <device name>, and I cannot find neither on your website nor on the CD-ROM the open source software used to build or modify the firmware.
>
> In conformance to the GPL license, you have to release the following sources:
>
> - complete toolchain that made the kernel and applications be compiled (gcc, binutils, libc)
> - tools to build a custom firmware (mksquashfs, mkcramfs ...)
> - kernel sources with patches to make it run on this specific hardware, this does not include binary drivers
>
> Thank you very much in advance for your answer.
>
> Best regards, <your name>

Using the SDK

Once the SDK is available, you are most likely not to be able to build a complete or functional firmware using it, but parts of it, like only the kernel, or only the root filesystem. Most manufacturers do not really care releasing a tool that do work every time you uncompress and use it.

You should anyway be able to use the following components:

- kernel sources with more or less functional patches for your hardware
- binary drivers linked or to be linked with the shipped kernel version
- packages of the toolchain used to compile the whole firmware: gcc, binutils, libc or uClibc
- binary tools to create a valid firmware image

Your work can be divided into the following tasks:

- create a clean patch of the hardware specific part of the linux kernel
- spot potential kernel GPL violations especially on netfilter and USB stack stuff
- make the binary drivers work, until there are open source drivers
- use standard a GNU toolchain to make working executables
- understand and write open source tools to generate a valid firmware image

Creating a hardware specific kernel patch

Most of the time, the kernel source that comes along with the SDK is not really clean, and is not a standard Linux version, it also has architecture specific fixes backported from the **CVS** or the **git** repository of the kernel development trees. Anyway, some parts can be easily isolated and used as a good start to make a vanilla kernel work your hardware.

Some directories are very likely to have local modifications needed to make your hardware be recognized and used under Linux. First of all, you need to find out the linux kernel version that is used by your hardware, this can be found by editing the **linux/Makefile** file.

```
head -5 linux-2.x.x/Makefile
VERSION = 2
PATCHLEVEL = x
SUBLEVEL = y
EXTRAVERSION = z
NAME=A fancy name
```

So now, you know that you have to download a standard kernel tarball at **kernel.org** that matches the version being used by your hardware.

Then you can create a **diff** file between the two trees, especially for the following directories:

```
diff -urN linux-2.x.x/arch/<sub architecture> linux-2.x.x-modified/arch/<sub architec
architecture.patch
diff -urN linux-2.x.x/include/ linux-2.x.x-modified/include > 02-includes.patch
diff -urN linux-2.x.x/drivers/ linux-2.x.x-modified/drivers > 03-drivers.patch
```

This will constitute a basic set of three patches that are very likely to contain any needed modifications that has been made to the stock Linux kernel to run on your specific device. Of course, the content produced by the **diff -urN** may not always be relevant, so that you have to clean up those patches to only let the ”must have” code into them.

The first patch will contain all the code that is needed by the board to be initialized at startup, as well as processor detection and other boot time specific fixes.

The second patch will contain all useful definitions for that board: addresses, kernel granularity, redefinitions, processor family and features ...

The third patch may contain drivers for: serial console, ethernet NIC, wireless NIC, USB NIC ... Most of the time this patch contains nothing else than ”glue” code that has been added to make the binary driver work with the Linux kernel. This code might not be useful if you plan on writing drivers from scratch for this hardware.

Using the device bootloader

The bootloader is the first program that is started right after your device has been powered on. This program, can be more or less sophisticated, some do let you do network booting, USB mass storage booting ... The bootloader is device and architecture specific, some bootloaders were designed to be universal such as RedBoot or U-Boot so that you can meet those loaders on totally different platforms and expect them to behave the same way.

If your device runs a proprietary operating system, you are very likely to deal with a proprietary boot loader as well. This may not always be a limitation, some proprietary bootloaders can even have source code available (i.e : Broadcom CFE).

According to the bootloader features, hacking on the device will be more or less easier. It is very probable that the bootloader, even exotic and rare, has a documentation somewhere over the Internet. In order to know what will be possible with your bootloader and the way you are going to hack the device, look over the following features :

- does the bootloader allow net booting via bootp/DHCP/NFS or tftp
- does the bootloader accept loading ELF binaries ?
- does the bootloader have a kernel/firmware size limitation ?
- does the bootloader expect a firmware format to be loaded with ?
- are the loaded files executed from RAM or flash ?

Net booting is something very convenient, because you will only have to set up network booting servers on your development station, and keep the original firmware on the device till you are sure you can replace it. This also prevents your device from being flashed, and potentially bricked every time you want to test a modification on the kernel/filesystem.

If your device needs to be flashed every time you load a firmware, the bootlader might only accept a specific firmware format to be loaded, so that you will have to understand the firmware format as well.

Making binary drivers work

As we have explained before, manufacturers do release binary drivers in their GPL tarball. When those drivers are statically linked into the kernel, they become GPL as well, fortunately or unfortunately, most of the drivers are not statically linked. This anyway lets you a chance to dynamically link the driver with the current kernel version, and try to make them work together.

This is one of the most tricky and grey part of the fully open source projects. Some drivers require few modifications to be working with your custom kernel, because they worked with an earlier kernel, and few modifications have been made to the kernel in-between those versions. This is for instance the case with

the binary driver of the Broadcom BCM43xx Wireless Chipsets, where only few differences were made to the network interface structures.

Some general principles can be applied no matter which kernel version is used in order to make binary drivers work with your custom kernel:

- turn on kernel debugging features such as:
  - CONFIG_DEBUG_KERNEL
  - CONFIG_DETECT_SOFTLOCKUP
  - CONFIG_DEBUG_KOBJECT
  - CONFIG_KALLSYMS
  - CONFIG_KALLSYMS_ALL
- link binary drivers when possible to the current kernel version
- try to load those binary drivers
- catch the lockups and understand them

Most of the time, loading binary drivers will fail, and generate a kernel oops. You can know the last symbol the binary drivers attempted to use, and see in the kernel headers file, if you do not have to move some structures field before or after that symbol in order to keep compatibily with both the binary driver and the stock kernel drivers.

Understanding the firmware format

You might want to understand the firmware format, even if you are not yet capable of running a custom firmware on your device, because this is sometimes a blocking part of the flashing process.

A firmware format is most of the time composed of the following fields:

- header, containing a firmware version and additional fields: Vendor, Hardware version ...
- CRC32 checksum on either the whole file or just part of it
- Binary and/or compressed kernel image
- Binary and/or compressed root filesystem image
- potential garbage

Once you have figured out how the firmware format is partitioned, you will have to write your own tool that produces valid firmware binaries. One thing to be very careful here is the endianness of either the machine that produces the binary firmware and the device that will be flashed using this binary firmware.

Writing a flash map driver

The flash map driver has an important role in making your custom firmware work because it is responsible of mapping the correct flash regions and associated rights to specific parts of the system such as: bootloader, kernel, user filesystem.

Writing your own flash map driver is not really a hard task once you know how your firmware image and flash is structured. You will find below a commented example that covers the case of the device where the bootloader can pass to the kernel its partition plan.

First of all, you need to make your flash map driver be visible in the kernel configuration options, this can be done by editing the file **linux/drivers/mtd/maps/Kconfig**:

```
config MTD_DEVICE_FLASH
        tristate "Device Flash device"
        depends on ARCHITECTURE && DEVICE
        help
         Flash memory access on DEVICE boards. Currently only works with
         Bootloader Foo and Bootloader Bar.
```

Then add your source file to the **linux/drivers/mtd/maps/Makefile**, so that it will be compiled along with the kernel.

```
obj-\$(CONFIG_MTD_DEVICE_FLASH)      += device-flash.o
```

You can then write the kernel driver itself, by creating a **linux/drivers/mtd/maps/device-flash.c** C source file.

```
// Includes that are required for the flash map driver to know of the prototypes:
#include <asm/io.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/mtd/map.h>
#include <linux/mtd/mtd.h>
#include <linux/mtd/partitions.h>
#include <linux/vmalloc.h>

// Put some flash map definitions here:
#define WINDOW_ADDR 0x1FC00000         /* Real address of the flash */
#define WINDOW_SIZE 0x400000                /* Size of flash */
#define BUSWIDTH 2                                  /* Buswidth */

static void __exit device_mtd_cleanup(void);

static struct mtd_info *device_mtd_info;

static struct map_info devicd_map = {
       .name = "device",
```

```
        .size = WINDOW_SIZE,
        .bankwidth = BUSWIDTH,
        .phys = WINDOW_ADDR,
};

static int __init device_mtd_init(void)
{
  // Display that we found a flash map device
        printk("device: 0x\%08x at 0x\%08x\n", WINDOW_SIZE, WINDOW_ADDR);
   // Remap the device address to a kernel address
        device_map.virt = ioremap(WINDOW_ADDR, WINDOW_SIZE);

        // If impossible to remap, exit with the EIO error
        if (!device_map.virt) {
                printk("device: Failed to ioremap\n");
                return -EIO;
        }

   // Initialize the device map
        simple_map_init(&device_map);

   /* MTD informations are closely linked to the flash map device
        you might also use "jedec_probe" "amd_probe" or "intel_probe" */
        device_mtd_info = do_map_probe("cfi_probe", &device_map);

  if (device_mtd_info) {
                device_mtd_info->owner = THIS_MODULE;

int parsed_nr_parts = 0;

// We try here to use the partition schema provided by the bootloader specific code
                        if (parsed_nr_parts == 0) {
                                int ret = parse_bootloader_partitions(device_mtd_info,
                                if (ret > 0) {
                                        part_type = "BootLoader";
                                        parsed_nr_parts = ret;
                                }
                        }

                        add_mtd_partitions(devicd_mtd_info, parsed_parts, parsed_nr_pa

                        return 0;
                }
        iounmap(device_map.virt);

        return -ENXIO;
}

// This function will make the driver clean up the MTD device mapping
static void __exit device_mtd_cleanup(void)
```

```
{
  // If we found a MTD device before
       if (device_mtd_info) {
   // Delete every partitions
                del_mtd_partitions(device_mtd_info);
   // Delete the associated map
                map_destroy(device_mtd_info);
       }

// If the virtual address is already in use
       if (device_map.virt) {
// Unmap the physical address to a kernel space address
                iounmap(device_map.virt);
// Reset the structure field
               device_map.virt = 0;
       }
}


// Macros that indicate which function is called on loading/unloading the module
module_init(device_mtd_init);
module_exit(device_mtd_cleanup);


// Macros defining license and author, parameters can be defined here too.
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Me, myself and I <memyselfandi@domain.tld");
```

2.4.3 Adding your target in OpenWrt

Once you spotted the key changes that were made to the Linux kernel to support your target, you will want to create a target in OpenWrt for your hardware. This can be useful to benefit from the toolchain that OpenWrt builds as well as the resulting user-space and kernel configuration options.

Provided that your target is already known to OpenWrt, it will be as simple as creating a `target/linux/board` directory where you will be creating the following directories and files.

Here for example, is a `target/linux/board/Makefile`:

```
1 #
2 # Copyright (C) 2009 OpenWrt.org
3 #
4 # This is free software, licensed under the GNU General Public License v2.
5 # See /LICENSE for more information.
6 #
7 include $(TOPDIR)/rules.mk
8
9 ARCH:=mips
```

```
10 BOARD:=board
11 BOARDNAME:=Eval board
12 FEATURES:=squashfs jffs2 pci usb
13
14 LINUX_VERSION:=2.6.27.10
15
16 include $(INCLUDE_DIR)/target.mk
17
18 DEFAULT_PACKAGES += hostapd-mini
19
20 define Target/Description
21         Build firmware images for Evaluation board
22 endef
23
24 $(eval $(call BuildTarget))
```

- `ARCH`
  The name of the architecture known by Linux and uClibc
- `BOARD`
  The name of your board that will be used as a package and build directory identifier
- `BOARDNAME`
  Expanded name that will appear in menuconfig
- `FEATURES`
  Set of features to build filesystem images, USB, PCI, VIDEO kernel support
- `LINUX_VERSION`
  Linux kernel version to use for this target
- `DEFAULT_PACKAGES`
  Set of packages to be built by default

A partial kernel configuration which is either named `config-default` or which matches the kernel version `config-2.6.x` should be present in `target/linux/board/`. This kernel configuration will only contain the relevant symbols to support your target and can be changed using `make kernel_menuconfig`.

To patch the kernel sources with the patches required to support your hardware, you will have to drop them in `patches` or in `patches-2.6.x` if there are specific changes between kernel versions. Additionnaly, if you want to avoid creating a patch that will create files, you can put those files into `files` or `files-2.6.x` with the same directory structure that the kernel uses (e.g: drivers/mtd/maps, arch/mips ..).

The build system will require you to create a `target/linux/board/image/Makefile`:

```
1 #
2 # Copyright (C) 2009 OpenWrt.org
```

```
3  #
4  # This is free software, licensed under the GNU General Public License v2.
5  # See /LICENSE for more information.
6  #
7  include $(TOPDIR)/rules.mk
8  include $(INCLUDE_DIR)/image.mk
9
10 define Image/BuildKernel
11         cp $(KDIR)/vmlinux.elf $(BIN_DIR)/openwrt-$(BOARD)-vmlinux.elf
12         gzip -9n -c $(KDIR)/vmlinux > $(KDIR)/vmlinux.bin.gz
13         $(STAGING_DIR_HOST)/bin/lzma e $(KDIR)/vmlinux $(KDIR)/vmlinux.bin.l7
14         dd if=$(KDIR)/vmlinux.bin.l7 of=$(BIN_DIR)/openwrt-$(BOARD)-vmlinux.lzma bs=65536 conv=
15         dd if=$(KDIR)/vmlinux.bin.gz of=$(BIN_DIR)/openwrt-$(BOARD)-vmlinux.gz bs=65536 conv=sy
16 endef
17
18 define Image/Build/squashfs
19     $(call prepare_generic_squashfs,$(KDIR)/root.squashfs)
20 endef
21
22 define Image/Build
23         $(call Image/Build/$(1))
24         dd if=$(KDIR)/root.$(1) of=$(BIN_DIR)/openwrt-$(BOARD)-root.$(1) bs=128k conv=sync
25
26         -$(STAGING_DIR_HOST)/bin/mkfwimage \
27                 -B XS2 -v XS2.ar2316.OpenWrt \
28                 -k $(BIN_DIR)/openwrt-$(BOARD)-vmlinux.lzma \
29                 -r $(BIN_DIR)/openwrt-$(BOARD)-root.$(1) \
30                 -o $(BIN_DIR)/openwrt-$(BOARD)-ubnt2-$(1).bin
31 endef
32
33 $(eval $(call BuildImage))
34
```

- `Image/BuildKernel`
  This template defines changes to be made to the ELF kernel file

- `Image/Build`
  This template defines the final changes to apply to the rootfs and kernel, either combined or separated firmware creation tools can be called here as well.

2.5 Debugging and debricking

Debugging hardware can be tricky especially when doing kernel and drivers development. It might become handy for you to add serial console to your device as well as using JTAG to debug your code.

2.5.1 Adding a serial port

Most routers come with an UART integrated into the System-on-chip and its pins are routed on the Printed Circuit Board to allow debugging, firmware replacement or serial device connection (like modems).

Finding an UART on a router is fairly easy since it only needs at least 4 signals (without modem signaling) to work : VCC, GND, TX and RX. Since your router is very likely to have its I/O pins working at 3.3V (TTL level), you will need a level shifter such as a Maxim MAX232 to change the level from 3.3V to your computer level which is usually at 12V.

To find out the serial console pins on the PCB, you will be looking for a populated or unpopulated 4-pin header, which can be far from the SoC (signals are relatively slow) and usually with tracks on the top or bottom layer of the PCB, and connected to the TX and RX.

Once found, you can easily check where is GND, which is connected to the same ground layer than the power connector. VCC should be fixed at 3.3V and connected to the supply layer, TX is also at 3.3V level but using a multimeter as an ohm-meter and showing an infinite value between TX and VCC pins will tell you about them being different signals (or not). RX and GND are by default at 0V, so using the same technique you can determine the remaining pins like this.

If you do not have a multimeter a simple trick that usually works is using a speaker or a LED to determine the 3.3V signals. Additionnaly most PCB designer will draw a square pad to indicate ping number 1.

Once found, just interface your level shifter with the device and the serial port on the PC on the other side. Most common baudrates for the off-the-shelf devices are 9600, 38400 and 115200 with 8-bits data, no parity, 1-bit stop.

2.5.2 JTAG

JTAG stands for Joint Test Action Group, which is an IEEE workgroup defining an electrical interface for integrated circuit testing and programming.

There is usually a JTAG automate integrated into your System-on-Chip or CPU which allows an external software, controlling the JTAG adapter to make it perform commands like reads and writes at arbitray locations. Additionnaly it can be useful to recover your devices if you erased the bootloader resident on the flash.

Different CPUs have different automates behavior and reset sequence, most likely you will find ARM and MIPS CPUs, both having their standard to allow controlling the CPU behavior using JTAG.

Finding JTAG connector on a PCB can be a little easier than finding the UART since most vendors leave those headers unpopulated after production. JTAG connectors are usually 12, 14, or 20-pins headers with one side of the connector having some signals at 3.3V and the other side being connected to GND.

2.6 Reporting bugs

2.6.1 Using the Trac ticket system

OpenWrt as an open source software opens its development to the community by having a publicly browseable subversion repository. The Trac software which comes along with a Subversion frontend, a Wiki and a ticket reporting system is used as an interface between developers, users and contributors in order to make the whole development process much easier and efficient.

We make distinction between two kinds of people within the Trac system:

- developers, able to report, close and fix tickets
- reporters, able to add a comment, patch, or request ticket status

Opening a ticket

A reporter might want to open a ticket for the following reasons:

- a bug affects a specific hardware and/or software and needs to be fixed
- a specific software package would be seen as part of the official OpenWrt repository
- a feature should be added or removed from OpenWrt

Regarding the kind of ticket that is open, a patch is welcome in those cases:

- new package to be included in OpenWrt
- fix for a bug that works for the reporter and has no known side effect
- new features that can be added by modifying existing OpenWrt files

Once the ticket is open, a developer will take care of it, if so, the ticket is marked as ”accepted” with the developer name. You can add comments at any time to the ticket, even when it is closed.

Closing a ticket

A ticket might be closed by a developer because:

- the problem is already fixed (wontfix)
- the problem described is not judged as valid, and comes along with an explanation why (invalid)
- the developers know that this bug will be fixed upstream (wontfix)

- the problem is very similar to something that has already been reported (duplicate)
- the problem cannot be reproduced by the developers (worksforme)

At the same time, the reporter may want to get the ticket closed since he is not longer able to trigger the bug, or found it invalid by himself.

When a ticket is closed by a developer and marked as ”fixed”, the comment contains the subversion changeset which corrects the bug.

2.7 Submitting patches

2.7.1 How to contribute

OpenWrt is constantly being improved. We'd like as many people to contribute to this as we can get. If you find a change useful, by all means try to get it incorporated into the project. This should improve OpenWrt and it should help carry your changes forward into future versions

This section tries to lay out a procedure to enable people to submit patches in a way that is most effective for all concerned.

It is important to do all these steps repeatedly:

- *listen* to what other people think.
- *talk* explaining what problem you are addressing and your proposed solution.
- *do* write useful patches including documentation.
- *test. test. test.*

2.7.2 Where to listen and talk

- google to find things related to your problem
- Mailing lists: http://lists.openwrt.org/
- Wiki: check the wiki: http://wiki.openwrt.org/OpenWrtDocs
- Forum: http://forum.openwrt.org/
- IRC: `irc.freenode.net`, channels `#openwrt` and `#openwrt-devel`
- TRAC: https://dev.openwrt.org/ the issue/bug/change tracking system

It is often best to document what you are doing before you do it. The process of documentation often exposes possible improvements. Keep your documentation up to date.

2.7.3 Patch Submission Process

1. Use git or svn to create a patch. Creating patches manually with *diff -urN* also works, but is usually unnecessary.
2. Send a mail to openwrt-devel@lists.openwrt.org with the following contents:
   (a) `[PATCH] <short description>` in the Subject, followed by:
   (b) (optional) a longer description of your patch in the message body
   (c) `Signed-off-by: Your name <your@email.address>`
   (d) Your actual patch, inline, not word wrapped or whitespace mangled.
3. Please read http://kerneltrap.org/Linux/Email_Clients_and_Patches to find out how to make sure your email client doesn't destroy your patch.
4. Please use your real name and email address in the `Signed-off-by` line, following the same guidelines as in the Linux Kernel patch submission guidelines
5. Example of a properly formatted patch submission: http://lists.openwrt.org/pipermail/openwrt-devel/2007-November/001334.html